A look inside behavioral synthesis
Michael Meredith
(04/08/2004 6:00 PM EDT)
URL: http://www.eetimes.com/showArticle.jhtml?articleID=18900783

Behavioral synthesis is an automated design process that interprets an algorithmic description of a desired behavior and creates hardware that implements that behavior. It is used as part of a behavioral design flow that promises to raise the level of abstraction of the design process. This can be shown to increase designer productivity and reduce the opportunity for error.

Starting with an algorithmic description in a high-level language, behavioral synthesis tools automatically create the cycle-by-cycle detail needed for hardware implementation. Most behavioral synthesis approaches leverage the existing logic synthesis toolset by creating a register transfer level (RTL) implementation from the algorithmic description.

This RTL is used directly in a conventional logic synthesis flow to create a gate-level implementation. These behavioral synthesis tools transform untimed or partially timed functional code into fully timed RTL implementations.

A behavioral design flow using behavioral synthesis lets the designer focus on the module functionality and the interconnect protocol. The design of the micro-architecture and the cycle-by-cycle timing are handled by the behavioral synthesis toolset.

A number of papers have been published describing various approaches to this problem. Results of using the behavioral synthesis approach on various hardware design projects have also been presented. Review of this literature indicates that designing at a higher level of abstraction using behavioral synthesis reduces the amount of code that must be developed by as much as two thirds. These users report overall reduction of design effort of 50% or more.

Differences between synthesizable RTL behavioral code

These advantages are made possible by distinct differences in the style of code that is written for RTL design, compared to that which is written for a behavioral design flow. The most visible difference is the level of abstraction of the design description.

The design description made possible by a behavioral synthesis design flow differs in a number of specific ways from that which is required for traditional logic synthesis. Logic synthesis uses an RTL description of the design. Behavioral synthesis uses a high-level untimed, or partially timed, functional description. Let us consider a few of the specific differences that are primarily responsible for the reduction in code size and the resulting productivity benefits of using a behavioral design flow.

Multi-cycle functionality

It is a fundamental characteristic of synthesizable RTL code that the complete functionality of each clocked process must be performed within a single clock cycle. Behavioral synthesis lifts this restriction. Clocked processes in synthesizable behavioral code may contain functionality that takes more than one clock cycle to execute.
The behavioral synthesis algorithms will create a schedule that determines how many clock cycles will be used. The behavioral synthesis tool automatically creates the finite state machine (FSM) that is required to implement this multi-cycle behavior in the generated RTL code.

In a traditional RTL design process, the designer is responsible for manually decomposing multi-cycle functionality into a set of single-cycle processes. Typically this entails the creation of multiple processes to implement the finite state machine, and the creation of processes for each operation and each output.

A behavioral synthesis tool performs this decomposition for the designer. The multi-cycle behavior can be expressed in a natural way in a single process leading to more efficient design specification and debug.

**Loops**

Most algorithms include looping structures. Traditional RTL design imposes severe restrictions on the use of loops, or prohibits them outright. Some RTL logic synthesis tools permit for loops with fixed loop indices only. The loop body is restricted to being executed in a single cycle. Parallel hardware is inferred for each loop iteration.

These restrictions require the designer to transform the algorithm into a multi-cycle FSM adding substantial complexity to the designer’s task. Behavioral design manages this complexity for the designer by permitting free use of loops. “While” loops and “for” loops with data-dependent loop indices are fully supported in a behavioral design flow. Loop termination constructs such as the C language “break” and “continue” keywords are permitted.

**Memory access**

In general, reading and writing to memories requires complex multi-cycle protocols. In RTL design these are implemented as explicit FSMs. Worse, these accesses must usually be incorporated in an already complex FSM implementing an algorithm.

Behavioral synthesis permits them to be represented in an intuitive way as simple array accesses. An array is declared in the native syntax of the behavioral language in use, tool directives are provided to control the mapping of the array to a physical memory element, and the array elements are referenced using the array indexing syntax of the language. The behavioral synthesis tool instantiates the memory element and connects it to the rest of the circuit. It also develops of the FSM for the memory access protocol and integrates this FSM with the rest of the algorithm.

**User comparison of RTL and behavioral flows**

A number of comparisons have been made in the literature between the traditional RTL-based logic synthesis flow and the behavioral design flow. In a paper presented at DesignCon in 1998, David Johnson and co-authors from Motorola cited a 50% to 60% reduction in design effort for behavioral design over RTL design in five design cases they studied.

In a Design Automation Conference paper titled Comparing RTL and Behavioral Design Methodologies in the Case of a 2M-Transistor ATM Shaper, Imed Mousa and co-authors performed parallel design efforts using each flow for the same design. They found substantial productivity and quality improvements using the behavioral
design flow, including a two-thirds reduction in lines of code and a two-thirds reduction in man-months to complete.

The identified causes of this improvement include:

- Combining wait and procedure calls. By encapsulating the I/O protocol in parameterized subroutines, a substantial reduction in the complexity and size of the code was achieved.
- Description of complex processes. The RTL design team split one part of the design into two modules because “designing a complex process in RTL is very tedious and error prone.” The behavioral design flow permitted this to be described in a more natural way using a single process. The resulting simplicity reduced the design effort. “The difference in number of lines is due to the fact that behavioral specification allows us to specify complex protocols in an algorithmic way mixing wait statements and control statements. In RTL, handshakes need to be specified as verbose FSMs,” the paper said.
- Mixing loop, if, and wait statements. “This is probably the most significant difference between RTL and behavioral coding styles.” RTL coding restrictions require these control structures to be implemented in more complex ways such as manually introducing an FSM.

A common characteristic of the designs considered in these papers is the relative dominance of data flow over control in the content of the algorithm. This has historically been the primary application of this technology. New behavioral synthesis techniques are emerging which are expanding the class of blocks to which this technology can be applied.

**Stages of the behavioral synthesis process**

The behavioral synthesis process consists of a number of activities. Various behavioral synthesis tools perform these activities in different orders using different algorithms. Some behavioral synthesis tools combine some of these activities or perform them iteratively to converge on the desired solution.

**Lexical processing**

Behavioral synthesis begins with an algorithmic description of the desired behavior expressed in a high-level language. Lexical processing parses the high-level language source code and transforms it into an internal representation. Lexical processing for behavioral synthesis is similar to that used in conventional high-level language compilation.

**Algorithm optimization**

Optimizations that can be performed on the algorithm itself include common subexpression elimination and constant folding. Many of these optimizations are commonly used in high-level language compilers or parallelizing compilers.

**Control/Dataflow analysis**

The inputs, outputs, and operations of the algorithm are identified, and the data dependencies between them are determined. The result of this process is usually a Control/Dataflow Graph (CDFG). This determines which values are needed prior to computation of other values. No concept of time exists in the CDFG.
Library processing
The RTL implementation produced by behavioral synthesis will depend on the capabilities and characteristics of the library parts available for the specific implementation technology to be used. Library processing reads the available libraries and determines the functional, timing, and area characteristics of the available parts.

Resource allocation
Resource allocation establishes a set of functional units that will be adequate to implement the design. In many behavioral synthesis systems, an initial resource allocation is performed and subsequently modified during scheduling and/or binding.

Scheduling
Scheduling introduces parallelism and the concept of time. It transforms the algorithm into an FSM representation. Using the data dependencies of the algorithm and the latencies of the functional units in the library, the operations of the algorithm are assigned to specific clock cycles. There are often many possible schedules. Directives that constrain the result with respect to latency, pipelining, and resource utilization will affect the schedule that is chosen.

Functional unit binding
Binding assigns the operations of the algorithm to specific instances of functional units from the library.

Register binding
In cases where values are produced in one clock cycle and consumed in another, these values must be stored in registers or memory. The register binding process allocates registers as needed and assigns each value to a physical register. Analysis of the lifetime of each data value can identify opportunities to use the same physical register to store different values at different times. This is done to reduce the size of the resulting design.

Output processing
The datapath and finite state machine resulting from all of the previous steps are written out as RTL source code in the target language. This code can be structured in a number of ways to optimize the downstream logic synthesis process or to enhance the readability of the code.

Example
Suppose we need to compute \(((a * b) + c) * (d * e)\) where each value is 8 bits. We can express this in the C language as:

```c
unsigned long
example_func(unsigned char a, unsigned char b, unsigned char c,

              unsigned char d, unsigned char e, unsigned char f)
{
    unsigned long y;
    y = ((a * b) + c) * (d * e);
    return y;
}
```
In order to build hardware to perform this computation, we will need to deal with I/O protocol, but it is useful to consider the algorithm in isolation to understand the behavioral synthesis process.

The control/dataflow analysis would construct a graph from this code as follows:

![Graph Diagram]

Assume our library contains the following functional units:

<table>
<thead>
<tr>
<th>Functional Unit</th>
<th>Delay</th>
<th>Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>8x8=16</td>
<td>2.78</td>
<td>4895.5</td>
</tr>
<tr>
<td>15+16=17</td>
<td>1.99</td>
<td>1440.3</td>
</tr>
<tr>
<td>20x20=40</td>
<td>5.99</td>
<td>27592.6</td>
</tr>
</tbody>
</table>

An initial allocation might be:

- Two 8x8=16 multipliers
- One 16+16=17 adder
- One 20x20=40 multiplier

Given this allocation, if the latency of this computation were constrained to be three cycles or less, the following trivial schedule can be constructed:

<table>
<thead>
<tr>
<th>Operator</th>
<th># Needed</th>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>8x8=16</td>
<td>2</td>
<td>a<em>b, d</em>e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16+15=17</td>
<td>1</td>
<td></td>
<td>t1+c</td>
<td></td>
</tr>
<tr>
<td>20x20=40</td>
<td>1</td>
<td></td>
<td>t2*t3</td>
<td></td>
</tr>
</tbody>
</table>

The scheduling algorithm should note that the d*e operation has “mobility” to be scheduled in either cycle 1 or cycle 2. It should then schedule it in cycle 2 in order to eliminate one 8x8 multiply operator.

<table>
<thead>
<tr>
<th>Operator</th>
<th># Needed</th>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>8x8=16</td>
<td>1</td>
<td>a*b</td>
<td>d*e</td>
<td></td>
</tr>
<tr>
<td>16+15=17</td>
<td>1</td>
<td>t1+c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20x20=40</td>
<td>1</td>
<td></td>
<td>t2*t3</td>
<td></td>
</tr>
</tbody>
</table>

The binding process now assigns these operators to specific instances of functional units. It may optimize this by noting that the 20x20 multiplier can perform the
function of the 8x8 multiplier. It may also select a smaller but slower adder if the timing permits.

<table>
<thead>
<tr>
<th>Operator</th>
<th># Needed</th>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>16+16=17</td>
<td>1</td>
<td>t1+c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20x20=400</td>
<td>1</td>
<td>a*b</td>
<td>d*e</td>
<td>t2*t3</td>
</tr>
</tbody>
</table>

This would imply an architecture like this:

![Architecture Diagram]

**Conclusion**

We have taken a look at the relatively new behavioral design methodology, and at the behavioral synthesis process that makes it possible. We have identified loops, memory access and multi-cycle behavior as the issues that primarily differentiate behavioral design from traditional RTL design.

We have reviewed results reported by users who experienced significant productivity improvements using behavioral synthesis compared to RTL implementation. We have discussed some of the processing stages that make up the behavioral synthesis process. Finally, we have examined how these stages might be applied in a specific example.

Behavioral synthesis is an emerging technology. As this is written, state-of-the-art behavioral synthesis tools are able to deliver significant productivity enhancement when applied to appropriate blocks within a design project. It is expected that as the capabilities of these tools expand, this approach will be become applicable to the more and more of the digital design process. The productivity improvements this enables will be necessary to face the challenges of designing upcoming systems that are expected to exceed one hundred million gates.